Hulk: Eliciting Malicious Behavior in Browser Extensions

Alexandros Kapravelos[◇] *Chris Grier*[†*] *Neha Chachra*[‡] *Christopher Kruegel*[◇]
Giovanni Vigna[◇] *Vern Paxson*[†*]
[◇]*UC Santa Barbara* [†]*UC Berkeley* [‡]*UC San Diego*
[*]*International Computer Science Institute*
{*kapravel, chris, vigna*}*@cs.ucsb.edu* {*grier, vern*}*@cs.berkeley.edu* *nchachra@cs.ucsd.edu*

Abstract

We present Hulk, a dynamic analysis system that detects malicious behavior in browser extensions by monitoring their execution and corresponding network activity. Hulk elicits malicious behavior in extensions in two ways. First, Hulk leverages *HoneyPages*, which are dynamic pages that adapt to an extension's expectations in web page structure and content. Second, Hulk employs a *fuzzer* to drive the numerous event handlers that modern extensions heavily rely upon. We analyzed 48K extensions from the Chrome Web store, driving each with over 1M URLs. We identify a number of malicious extensions, including one with 5.5 million affected users, stressing the risks that extensions pose for today's web security ecosystem, and the need to further strengthen browser security to protect user data and privacy.

1 Introduction

All major web browsers today support broad extension ecosystems that allow third parties to install a wide range of modified behavior or additional functionality. Internet Explorer has binary add-ons (Browser Helper Objects), while Firefox, Chrome, Opera, and Safari support JavaScript-based extensions. Some browsers have online web stores to distribute extensions to users. For example, the most popular extension in Chrome's Web Store, AdBlock, has over 10 million users. Other popular extensions serve a variety of functions, such as preserving privacy, changing the aesthetics of the browser's UI, or integrating with web services such as Google Translate.

The amount of critical and private data that web browsers mediate continues to increase, and naturally this data has become a target for criminals. In addition, the web's advertising ecosystem offers opportunities to profit by manipulating a user's everyday browsing behavior. As a result, malicious browser extensions have become a new threat, as criminals realize the potential to monetize a victim's web browsing session and readily access web-related content and private data.

Our work examines extensions for Google Chrome that are designed with malicious intent—a threat distinct from that posed by attackers exploiting bugs in benign extensions, which has seen prior study [6, 5]. Extensions for Google Chrome are primarily distributed through the Chrome Web Store.[1] Like app stores for other platforms, such as Android or iOS, inherent risks arise when downloading and executing programs from untrusted sources. Reports have documented not only malicious extensions [27], but miscreants *purchasing* extensions (and thereby access to their userbases via update mechanisms) to add malicious functionality [2, 25]. In addition to the web store, extensions can also be directly installed by users and other programs. Installed by a process called *sideloading*, these extensions pose a recognized risk that browser vendors have attempted to prevent through modifications to the browser [22]. Sideloaded extensions are especially problematic since they can be installed without user knowledge, and are not subject to review by a web store. Despite efforts to stifle sideloaded extensions, they remain a significant problem [12].

In this paper we present Hulk, a tool for detecting malicious behavior in Google Chrome extensions. Hulk relies on dynamic execution of extensions and uses several techniques to trigger malicious functionality during execution. One technique we developed to elicit malicious behavior is the use of *HoneyPages*: specially-crafted web pages designed to satisfy the structural conditions that trigger a given extension. We interpose on all queries and modifications to the DOM tree of the HoneyPage to automatically create elements and mimic DOM tree structures for extensions on the fly. Using this technique, we can readily observe malicious behavior that inserts new `iframe` or `div` elements.

In addition, we built a fuzzer to drive the execution

[1] https://chrome.google.com/webstore/category/extensions

of event handlers registered by extensions. In our experiments, we use the fuzzer to trigger all event handlers associated with web requests, exercising each with 1 million URLs. Although we undertook extensive efforts to trigger malicious behavior, the possibility remains that Hulk lacks the mechanisms to satisfy all of the conditions necessary for eliciting an extension's malicious behavior.

Our analysis of 48,332 Chrome extensions found that malicious extensions pose a serious threat to users. By developing a set of rules that label execution logs from Hulk, we identified 130 malicious extensions and 4,712 "suspicious" extensions, most of which appear in the Chrome Web Store. Several large classes of malicious behavior appear within our set of extensions: affiliate fraud, credential theft, ad injection or replacement, and social network abuse. In one case, an extension performing ad replacement had nearly 2 million users, similar in size to some of the largest botnets.

In summary, we frame our contributions as follows:

- We present Hulk, a system to perform dynamic analysis for Chrome extensions.
- We demonstrate the effectiveness of HoneyPages and event handler fuzzing to elicit malicious behavior in browser extensions.
- We perform the first broad study of malicious Chrome extensions.
- We characterize several classes of malicious Chrome extensions, some with very large footprints (up to 5.5M installations) and propose solutions to eliminate entire classes of malicious behavior.

2 Background

We begin by reviewing the Google Chrome extension model and the opportunities this model provides to malicious extensions.

2.1 Chrome Extension Composition

Google Chrome supports extensions written in JavaScript and HTML (distributed as a single `zip` file). A small number of extensions also include binary code plugins, although these are subject to a manual security review process [15]. Each extension contains a (mandatory) manifest that, along with other extension parameters, describes the permissions the extension uses and the list of resources that the browser should load.

The permission system is designed in the spirit of least privilege, with the goal of limiting the resources available to an extension in case it has exploitable vulnerabilities [5]. The threat model does not attempt to address malicious extensions accessing sensitive content or performing other actions. The permission system determines which sites an extension can access, the allowed API calls, and the use of binary plugins. We describe relevant parts of the permission system later in this section. See Barth et al. for a more detailed description of Chrome's extension architecture [5].

2.2 Installing Extensions

The Chrome Web Store is the official means for users to find and install extensions. The web store is similar to other app stores, such as those for iOS and Android, in that developers create extensions and upload them to the store for users to download. Extension developers can also push out updates without requiring any action by the end-user.

In addition to the Chrome Web Store, extensions can also be installed manually by a user or an external program. We refer to the installation of extensions outside the web store as *sideloading*. Chrome version 25 (released February, 2013) included changes to prevent silent installation of Chrome extensions and require that the user indicate consent for installation [22]. In May, 2014, Chrome took further steps to prevent sideloading by requiring *all* installed extensions to be hosted in the Chrome Web Store [18]. While these changes increase the difficulty of sideloading, it is still possible for programs to force silent installation of extensions, since the attacker already has control of the machine. For our study we obtained a set of extensions that are sideloaded into Chrome by other Windows programs, many of which are known malware.

2.3 Extension Permissions

Permissions. Chrome requires extensions to list the permissions needed to access the different parts of the extension API. For example, Figure 1 shows a portion of a manifest file requesting permission to access the `webRequest` and `cookies` API. The `webRequest` permission allows the extension to "observe and analyze traffic and to intercept, block, or modify requests in-flight" by allowing the extension to register callbacks associated with different parts of the HTTP stack [15]. Similarly, the cookies API allows the extension to get, set, and be notified of changes to cookies.

The extension API permissions operate in conjunction with the optional *host permissions*, which limit the API permissions to access resources only for the specified URLs. For example, in Figure 1 the extension requests host permissions for `https://www.google.com/`, which allows it to access `cookies` and `webRequest` APIs for the specified domains. Host permissions also support wildcarding

```
...
"permissions": [
  "cookies",
  "webRequest",
  "*://*.facebook.com/",
  "https://www.google.com/"
],
...
"content_scripts": [
  {
    "matches": ["http://www.yahoo.com/*"],
    "js": ["jquery.js", "myscript.js"]
  }
],
...
"background": {
  "scripts": ["background.js"]
},
...
"content_security_policy": "script-src 'self'
      http://www.foo.com 'unsafe-eval';"
...
```

Figure 1: Example of a manifest that shows API permissions for two hosts, followed by content scripts that run on `http://www.yahoo.com`, followed by a background script that runs on all pages. Finally, the CSP specifies the ability to include and `eval` scripts in the extension from `foo.com`.

URLs. In Figure 1, the extension requests access to `*://*.facebook.com`. This permission allows for access to all subdomains of `facebook.com` requested via any URL scheme. In addition to wildcards, the special token `<all_urls>` matches any URL.

Besides the permissions described above, we found that extensions request a variety of other permissions. In Section 4 we summarize the permissions requested for all of the extensions we examined, and we discuss the permissions relevant to various types of abuse in Section 5. Other resources provide a thorough analysis of the Chrome permission system [5, 6].

Content Scripts. In addition to permissions for accessing various resources associated with a page, extensions can also specify a list of `content_scripts` to indicate JavaScript files that will run inside of the web page. Figure 1 shows an example of including two JavaScript files, `jquery.js` and `myscript.js` that will be run in the context of the page for any URLs matching the specified URL patterns (all pages on `http://www.yahoo.com/` in this example). Inside of each JavaScript file the author can include further logic to decide if and when to execute.

The ability to run in the context of a page is a powerful feature. Once a content script executes, any resulting actions become indistinguishable from actions performed by JavaScript provided by the web server. Not only can the scripts modify the DOM tree or other scripts, but they can also issue authenticated web requests (such as POST with proper cookies).

Background Pages. Besides the content scripts that allow an extension to interact with a given page, Chrome also allows extensions to run scripts in a "background page". Figure 1 shows an example manifest file that specifies `background.js` as a background page. Background pages often contain the logic and state an extension needs for the entirety of the browser session and do not have any visibility to the user. For example, an extension requesting `webRequest` permissions may use the background script to attach a listener to read outgoing requests using the `chrome.webRequest.onBeforeRequest.addListener()` call. After filtering on the *host permissions*, Chrome will send the extension a notification for every outgoing request. We detail further examples in the context of the extensions in the following sections.

Content Security Policy. In general, servers can specify a Content Security Policy (CSP) header that the browser uses to determine the sources from which it can include objects on the page. CSP can also specify other options, such as whether to allow the page to perform an `eval` or to embed inline JavaScript [29]. Extensions can use the same syntax to express their CSP in the manifest file. For example, an extension that wishes to include source from `foo.com` and to execute `eval` can specify its CSP as shown in Figure 1.

3 Architecture

In this section, we describe the architecture of Hulk, our dynamic analysis system that identifies malicious behavior in Chrome extensions. Hulk dynamically loads extensions in a monitored environment and observes the interaction of extensions with the loaded web pages. Using a set of heuristics to identify potentially dangerous behavior, it labels extensions as *malicious*, *suspicious*, or *benign*. In the rest of this section we describe how Hulk works and the challenges that arise in analyzing browser extensions.

3.1 Profiling Extensions

At the core of our dynamic analysis system is an instrumented browser and extension loader that enables us to automatically install extensions and instrument activity during web browsing. Our monitoring hooks collect data

from multiple vantage points within Hulk as it visits web pages and triggers a range of extension behavior.

URL Extraction. Before we dynamically analyze an extension we need to ensure that we can trigger the extension's functionality. Most extensions interact with the content of web pages, so we need to choose which URLs to load for our analysis. To this end, we use three sources of URLs: the manifest, the source code, and a list of popular sites. First, using the manifest file of the extension we construct valid URLs that match the permissions and content scripts specified. In some cases, the host permissions of an extension are restrictive—for example, `https://*.facebook.com`—so we can generate URLs that will match the pattern. It is more difficult to pick URLs to visit in cases where the extension requests host permissions on all URLs (Section 2.3), because the malicious behavior may only trigger on a small subset of sites. Therefore, we search the source code for any static URLs and visit those as well. Finally, for every extension we also visit a set of popular sites targeted by malicious extensions. We constantly strive to improve this list as we detect malicious extensions attacking particular domains. We however note that although we use multiple sources of URLs to determine the appropriate pages to visit, our approach is not complete; we discuss the limitations further in Section 7.

HoneyPages. Some extensions activate based on the content of a web page instead of the URL. To analyze such extensions we use specially crafted pages that attempt to satisfy the conditions that an extension looks for on a page before performing an action. We call these *HoneyPages*. HoneyPages contain JavaScript functions that overload built-in functions that query the DOM tree of the web page. As a result, when an extension queries for the presence of a specific element we can automatically create it and insert it into the page. For example, if the extension queries an `iframe` DOM element with the intention to alter it, then our HoneyPage will create an `iframe` element, inject it in the DOM tree, and return it to the extension.

HoneyPages enable us to supplement the URL extraction phase and dynamically create an environment for the extension to perform as many actions as it needs. The on-demand nature of a HoneyPage does not restrict us to a specific DOM tree structure, but enables us to determine what an extension looks for in a page during execution, since we can record all interactions within a HoneyPage. By using HoneyPages we can better understand how the extension will behave on arbitrary pages that are otherwise difficult to generate prior to analysis.

3.2 Event-Based Execution

The Chrome browser offers to extensions an event-based model to register callbacks that respond to certain browser-level events. For example, extensions use the `chrome.webRequest.onBeforeRequest` callback to intercept all outgoing HTTP requests from the browser. HoneyPages will not trigger callbacks for network events that require special properties, such as a specific URL or HTTP header. Therefore, we complement HoneyPages with event handler fuzzing. Specifically, we invoke all event callbacks that an extension registers in the `chrome.webRequest` API with mock event objects. We point to a HoneyPage loaded in the active tab while invoking the callbacks, enabling us to monitor the changes that the extension attempts to make on that page. Our approach allows us to test for every extension the extension's callbacks on the top 1 million Alexa domains in under 10 seconds on average.

3.2.1 Monitoring Hooks

Browser Extension API. Depending on the permissions included in the manifest (Section 2.3), an extension can use the Chrome extension API to perform actions not available to JavaScript running in a web page. As such, monitoring the extension API captures a subset of the total JavaScript activity that results from an extension, but gives us a detailed picture of what the extension attempts to do. For example, we monitor the extension API and log if the extension registers a callback to intercept all HTTP requests performed by the browser, and then track the changes that the extension makes to the HTTP requests. To do this, we leverage the current logging infrastructure offered by Chrome for monitoring the activity of extensions. We build upon the JavaScript function call logging provided by the browser to identify malicious behavior, such as tampering of security-related HTTP headers.

Content Scripts. We intercept and log all additional code introduced by the extension in the context of the visited page. Doing so provides a more complete picture of the extension's functionality, since it can include remote scripts from arbitrary locations and inject them into the page. Remote scripts can compromise the page's security similar to third-party JavaScript libraries [23], and make the analysis of the extension more difficult. Using remote scripts gives miscreants the ability to blacklist IP addresses of our analysis system (i.e., cloaking [17, 28]) or return code without the malicious components. Remote JavaScript inclusion also renders static analysis on the extension's code fundamentally incomplete since parts of the extension's codebase are not available until execution.

Network Logging. We use a transparent proxy that intercepts all browser HTTP and DNS traffic to log the requests made during extension execution. A browser extension has a set of files available as resources loaded by the browser, and it can also download and execute content from the web. Since the URLs retrieved can be computed at runtime, monitoring the network activity of the extension is critical for a complete analysis of its source code and included components. In addition to identifying remote content, we log all domains contacted by monitoring the DNS requests generated by the browser. Doing so enables us to identify extensions that contact nonexistent domains, which can occur because the extension is no longer operational or up-to-date. In these cases, our analysis was necessarily incomplete, since when the domain was active the extension could have fetched more remote code from it.

3.3 Detecting Malicious Behavior

As described in the previous section, our dynamic analysis system can provide detailed information about all browser and extension activity performed while visiting web pages. We combine this data to label the extension as either *benign*, *suspicious*, or *malicious* by applying a set of labeling heuristics based on the behavior. Labeling an extension as malicious indicates we identified behavior harmful to the user. Suspicious indicates the presence of potentially harmful actions or exposing the user to new risks, but without certainty that these represent malicious actions. Finally, when we do not find any suspicious activity, we label the extension as benign.

3.3.1 JavaScript Attributes

We use our monitoring modules described in Section 3.2.1 to identify malicious JavaScript execution. Below we detail actions that we consider malicious or suspicious in our post-processing analysis.

Extension API. As described earlier, Chrome's extension API offers privileged access to additional functionality of the browser besides native JavaScript, using permissions specified in the manifest file. While there are benign uses for every permission, we found several extensions that abuse the API. Specifically, for reasons described below, we consider the following actions available only through the extension API as malicious: uninstalling other extensions, preventing uninstallation of the current extension, and manipulating HTTP headers.

We consider uninstalling other extensions as malicious because some extensions uninstall cleaner extensions, such as the extension Facebook created to remove harmful extensions on its blacklist.[2] We detect this behavior by monitoring the `chrome.management.uninstall` API calls. To avoid false positives, we can differentiate cleaners from malicious extensions because, to the best of our knowledge, cleaners operate in a different fashion than Antivirus does: they clean up malicious extensions and then remove themselves from the browser. This differs from the behavior of malicious extensions, which remain persistent on the system.

Besides attempting to uninstall other extensions, malicious extensions often prevent the user from uninstalling the extension itself. More specifically, we found extensions that prevent the user from opening Chrome's extension configuration page where a user can conveniently uninstall any extension. To prevent uninstallation, malicious extensions interfere with tabs that point to the extension configuration page, `chrome://extensions`, either by replacing the URL with a different one, or by removing the tab completely. For analysis, we load a tab with `chrome://extensions` in the browser during our dynamic analysis and monitor any interactions to identify such behavior.

Lastly, using callbacks in the `webRequest` API, a malicious extension can manipulate HTTP headers. Extensions can use the `webRequest` API to effectively perform a man-in-the-middle attack on HTTP requests and responses before they are handled by the browser. This behavior is often malicious (or at least dangerous) since we found extensions that remove security-related headers, such as Content-Security-Policy or X-Frame-Options, through the use of callbacks such as `webRequest.onHeadersReceived` and `webRequestInterval.eventHandled`. By monitoring the use of this API, we can log events that reveal state of HTTP headers before and after the request. Upon manipulation of any security-related headers, we label the extension as malicious.

Interaction with visited pages. In addition to the extension API, we also monitor an extension's use of content scripts to modify web content loaded in the browser. In our analysis, we flag two kinds of interaction: sensitive information theft as malicious and injection of remote JavaScript content as suspicious.

There are many ways an extension can steal personal information from the user. For example, it can act as a JavaScript-based keylogger by intercepting all keystrokes on a page. Extensions can also access form data, such as a password field, before it is encrypted and sent over the network. Finally, extensions can also steal sensitive information from third parties by accessing sites with which the user has a valid session, and ei-

[2] `https://chrome.google.com/webstore/detail/facebook-malicious-extens/mhkafbldkepdhhjpmedkngigkjjknoa`

ther issuing requests to exfiltrate data, or simply stealing valid authentication tokens.

We label any extension that injects remote JavaScript content into a web page as suspicious. We define this activity as adding a `script` element with a `src` attribute pointing to a domain that is different from the one of the web page. Including these scripts complicates analysis since the JavaScript content can change without any corresponding change in the extension. We have observed changes to JavaScript files that substantially alter the functionality of an extension, possibly due to a server compromise.

3.3.2 Network Level

By monitoring network requests, including DNS lookups and HTTP requests, we identify other types of suspicious/malicious behavior. Using a manual analysis of network logs we have identified two attributes that indicate malicious or suspicious behavior: request errors and modification of HTTP requests. To detect HTTP modifications, we examine if the network response that we observe on the wire differs from the network response finally processed by the browser.

As we discussed earlier, the extension API offers callbacks to give extensions the ability to intercept and manipulate web requests. Not only can extensions drop security-related headers, but extensions can change or add parameters in URLs before the HTTP request is sent. We find such suspicious behavior common, especially among extensions that request permissions on shopping-related sites such as Amazon, EBay, and others. In these cases, the extension adds parameters to the URL that indicate that the site should credit a particular affiliate for any resulting sales. We discuss this behavior in more detail in Section 5. At the network level, we have the complete view of how the requests originally appeared. We combine that knowledge with our `chrome.*` API monitoring to identify the exact changes made to the request.

We also look for errors during domain name resolution to identify extensions that contact domains since taken down. As with drive-by downloads, we expect that malicious code dynamically loaded into an extension will eventually become blacklisted. In such cases, the extension will fail to introduce more code during its execution. We detect this behavior and mark it as suspicious.

3.4 Injected Content Analysis

A Chrome extension can also manipulate the visited pages of the browser by injecting a content script. The injected script runs in the context of the visited page and thus has full access to its DOM tree. The injected code can vary significantly, and, with the dynamic nature of JavaScript, can prove difficult to analyze statically. The use of HoneyPages enables us to understand the injected code's full intentions. Instead of trying to infer what the code will do, we actually run it to observe its effects on the DOM tree and classify it accordingly. For example, if the injected code looks for a form field with the name "password," we classify it as malicious, since it can potentially hijack the user's credentials on the page. Another example concerns injecting additional code, where the injected code is part of a two-stage process that fetches yet more code from the web and dynamically executes it in the context of the visited page. By relying on HoneyPages to understand the code's intentions by the effect that the code has on a given page, we obtain a more precise view of what the code attempts do than we can using only static analysis.

Analysis result	**Count**
Malicious	130
Suspicious	4,712
Benign	43,490
Total	48,332

Table 1: Classification distribution of extensions.

Detection class	**Count**
[s] Injects dynamic JavaScript	2,672
[s] Produces HTTP 4xx errors	2,322
[s] Evals with input >128 chars long	451
[m] Prevents extension uninstall	56
[m] Steals password from form	39
[s] Performs requests to non-existent domain	26
[m] Contains keylogging functionality	23
[m] Injects security-related HTTP header	11
[m] Steals email address from form	10
[m] Uninstalls extensions	8

Table 2: Distribution of detected suspicious/malicious behavior from analyzed extensions. Notice that an extension might have more than one detections and that we mark with [m] detections classified as malicious and with [s] detections classified as suspicious.

4 Results

To evaluate Hulk we use two sources of extensions: the official Chrome Web Store (totaling 47,940 extensions), and extensions sideloaded by binaries. We obtained the latter based on binaries executed in Anubis [1], which, after removing a large number of duplicates, resulted in

Rank	Top 10 types of permissions	# ext.
1	tabs	16,787
2	notifications	12,011
3	unlimitedStorage	9,424
4	storage	5,725
5	contextMenus	4,774
6	cookies	2,872
7	webRequest	2,849
8	webRequestBlocking	2,102
9	webNavigation	1,623
10	management	1,533

Table 3: The top 10 permissions found in the manifest files for all extensions we ran. Extensions can include more than one permission.

a set of 392 unique extensions. As shown in Table 1, in total we analyzed 48,332 distinct extensions, of which Hulk labeled 130 as *malicious* and 4,712 as *suspicious*. Table 2 summarizes all of the detected behaviors, which we analyze in more detail in the following sections.

4.1 Permissions Used

In this section we characterize the extensions we executed by identifying the most popular permissions, content scripts, and API calls that they performed.

Permissions. Table 3 shows the top 10 permissions from 30,392 unique extensions that use the Chrome Extension API (excluding the host permissions). The most commonly used, the `tabs` permission, allows an extension to interact with the browser's tabs, including navigating a tab to a specified URL and registering callbacks to react to changes in the address bar. The second most popular permission, `notifications`, allows an extension to generate custom notifications that alert the user. The `storage` and `unlimitedStorage` permissions allow storing of permanent data in the user's browser. The `contextMenus` permission allows an extension to add additional items on the context menu of the browser. Context menus appear when the user right clicks on a page. To manipulate the browser's cookies, an extension needs to ask for the `cookies` permission. The permissions `webRequest`, `webRequestBlocking` and `webNavigation` allow an extension to inspect, intercept, block, or modify web requests from the browser. Finally, an extension can get a list of other extensions installed in the browser—and even disable or unistall them—with the `management` permission.

We also computed permission statistics independently for the set of benign extensions and the set of malicious or suspicious ones. To our surprise, we found

Rank	Top 25 hosts in permissions	# ext.
1	http://*/*	7,319
2	https://*/*	6,395
3	<all_urls>	2,044
4	http://*/	1,126
5	*://*/*	1,025
6	https://*/	665
7	www.flashgame90.com/Default.aspx	224
8	https://api.twitter.com/	200
9	http://localhost/*	161
10	http://127.0.0.1/*	133
11	https://secure.flickr.com/	95
12	*://*.facebook.com/*	91
13	*://*/	89
14	https://www.facebook.com/*	82
15	http://vk.com/*	77
16	http://*.facebook.com/*	77
17	https://mail.google.com/*	71
18	https://*.facebook.com/*	70
19	http://*.google.com/	68
20	https://www.google-analytics.com/	67
21	https://mail.google.com/	64
22	https://*.google.com/	62
23	https://twitter.com/*	61
24	https://www.googleapis.com/	60
25	google.com/accounts/OAuthGetAcc[..]	56

Table 4: The top 25 host permissions used by extensions. Extensions can include more than one host permission per manifest.

that permissions for benign extensions do not differ significantly from permissions requested by malicious/suspicious ones, indicating that often attackers do not need to target different APIs to perform their attacks; maliciousness instead manifests in the way they use the API.

We found 18,313 extensions that use host permissions to restrict on which pages the extension can use the privileged `chrome.*` API. Table 4 shows the top 25 hosts appearing in host permissions. As seen in the table, extensions typically request broad permissions using wildcards in URL patterns. In addition these, we examined the hosts that extensions specified as targets for injecting content scripts, per Table 5, finding similar broad declarations. In practice, extension authors often use content scripts and host permissions in an unrestricted fashion.

API calls. Table 6 shows the top 15 Chrome Extension API calls made during by extensions during our experiments. There are several measurement artifacts introduced by our methodology. To load an extension for testing, we install the extension on a clean

Rank	Top 25 hosts in content_scripts	# ext.
1	http://*/*	12,472
2	https://*/*	10,864
3	<all_urls>	4,795
4	*://*/*	1,536
5	https://www.facebook.com/*	520
6	*://*.facebook.com/*	510
7	https://mail.google.com/*	458
8	http://www.facebook.com/*	433
9	https://*.facebook.com/*	344
10	http://*.facebook.com/*	320
11	file://*/*	315
12	https://twitter.com/*	303
13	http://mail.google.com/*	273
14	*://pages.brandthunder.com/[..]	265
15	https://plus.google.com/*	261
16	ftp://*/*	246
17	http://vk.com/*	227
18	http://www.youtube.com/*	211
19	file:///*	207
20	*://mail.google.com/*	189
21	http://twitter.com/*	179
22	*://www.facebook.com/*	178
23	http://ak.imgfarm.com/images[..]	177
24	*://*.reddit.com/*	164
25	https://vk.com/*	164

Table 5: The top 25 hosts used in extensions' content script permissions.

browser each time we start an analysis. This causes `runtime.onInstalled` to appear in every analysis independent of the extension's activities. We also open the `chrome://extensions` tab from inside the extension to determine if the extension interferes with the management of extensions. This causes Hulk to record a large number of `tabs.create` calls. In Table 6 the `tabs` API is by far the most used API, which matches the popularity of `tabs` permissions observed in Table 3.

4.2 Network Level

Using network activity alone we identified 24 malicious extensions. These extensions were labeled as malicious by Hulk because they tampered with security-related HTTP headers. By removing HTTP response headers like *Content-Security-Policy*, the malicious extensions can inject JavaScript into pages that specifically do not allow scripts from external sources (according to the CSP policies provided by the web server). For example, Hulk found multiple variants of an active extension on the Chrome Web Store targeting users that seek to cheat in

Rank	Top 15 chrome.* APIs called	# calls
1	runtime.onInstalled	182,476
2	webRequestInternal.eventHandled	57,466
3	tabs.getAllInWindow	49,312
4	tabs.onUpdated	32,354
5	tabs.create	25,947
6	i18n.getMessage	13,549
7	webRequest.onBeforeSendHeaders	13,213
8	runtime.connect	13,004
9	extension.getURL	11,942
10	storage.get	10,178
11	contextMenus.create	7,816
12	tabs.get	6,970
13	webRequest.onBeforeRequest	6,168
14	runtime.sendMessage	5,847
15	extension.sendRequest	5,454

Table 6: The top 15 chrome.* APIs called by extensions during dynamic analysis.

online games; these extensions, generally going by the name "*Cheat in your favorite games*", affect over 20K users.

During our experiments we encountered cases where our analysis could not obtain the full set of information needed to make a decision regarding the maliciousness of an analyzed extension. This problem arose due extensions performing HTTP requests that either returned errors, such as an HTTP 404 responses, or having domain names that no longer resolved. In such cases, given our inability to exercise the extension's full set of capabilities, and because the failed requests might correspond to fetching additional code, we mark these extensions as suspicious.

4.3 Extensions Management

Using signals tailored to detect the manipulation of the `chrome://extensions` page (as described in Section 3.3), we found several extensions on the Chrome Web Store that prevent uninstallation. Two of of these extensions claim to be video players (each with thousands of user) and completely replace Chrome's extensions management with a page that prevents users from uninstalling them. These are "HD Video Player" with 7,173 users and "SmartScreen Video Plugin" with 11,012 users. These signals also generated a false positive: the "No Tab Left Behind" extension (with only 8 users) allows only one tab at a time to be open. Thus, during our execution this extension prevented us from opening the extension settings tab.

4.4 Code Injection

Code injection was the most commonly detected "suspicious" feature in our dataset. In principle injection need not occur at all, since Chrome extensions can come packaged with all the code needed to operate. In total, we found more than 3,000 extensions that dynamically introduced remotely-retrieved code either through script injections or by evoking `eval`. As we noted earlier, using remote code renders static analysis on the extension's code fundamentally incomplete. However, Hulk can identify code injections and pinpoint the remote locations from which an extension fetches code. Although not necessarily malicious, we found many cases of dangerous code injection. For example, our system identified an extension named "Bang5TaoShopping assistant" from the Chrome Web Store that has been installed in 5.6 million (!) browsers and injects code into every visited page. Several extensions perform this same activity, while others insert tracking pixels for similar purposes. One instance sends cleartext HTTP request to a server controlled by the extension that encodes the URL visited by the user along with a unique identifier, leaking users browsing behavior and thus compromising their privacy.

5 Profiting from Maliciousness

In this section, we discuss five categories of malicious behavior in extensions, and describe their characteristics and the methods they employ to carry out their goals. We base each of these categories on examples we found in our feeds. When the extension is available on the Chrome Web Store, we also when possible include the number of users prior to reporting the extension to Google for review.

We have reported to Google any extension that performs behavior that is clearly abusive or malicious, and several of our reports have lead to removals of extensions from the web store.

5.1 Ad Manipulation

Advertisement manipulation falls in a grey area in that it does not subvert the user, but rather manipulates an external ecosystem. Replacing ads might appear benign to end users, but removes the potential for monetary credit for website owners (publishers) and instead fraudulently credits the extension owner. We include in this category the addition of new ads as well as the replacement of existing ads or identifiers. We find a range of behaviors in extensions, such as replacing banner ads with different identically-sized banners; inserting banners and text ads into well-known sites (such as Wikipedia); changing affiliate IDs for ads; or simply overlaying ads on top of

```
"content_scripts": [{
  "matches": ["http://*/*", "https://*/*"],
  "js": ["js/content.js"]
}],
"permissions": ["http://*/*",
  "https://*/*", "tabs"],
```

Figure 2: Permission-related JSON from the manifest file of an extension performing ad replacement.

content. Each instance aims to profit from impressions or clicks on the substituted advertisements.

As one striking example of ad manipulation we found an extension on the Chrome Web Store that had 1.8M users at the time we detected it. The extension, named "SimilarSites Pro" used primarily unobfuscated JS to perform benign functionality as advertised on the Chrome Web Store; however, it also inserted a script element into the content of web pages that downloads another, fully-obfuscated script (using `eval` and `unescape`) from a web server. At the time of analysis, this script contained a large conditional block that looked for `iframe` elements of particular sizes, such as 728x90 pixels, and replaced them with new banners of the same size. Since our first analysis, we have seen several new versions of the script available from the same URL. In addition, the extension contains a blacklist of sites and meta keywords where it should *not* change the banners, which appears due to many ad networks prohibiting the display of their ads on porn sites.

We find the same JavaScript included in five other extensions from the Chrome Web Store, as well as one sideloaded extension. Based on manual analysis, these extensions are primarily produced by a single company called "SimilarGroup" that engages in dubious behavior through the Chrome Web Store.

To perform banner replacement, the extension requests the permissions shown in Figure 2. Such exceptionally wide permissions are not uncommon [6]. Therefore, their presence alone provides little insight into the functionality of the extension. The most significant permission in Figure 2 is the broad use of content scripts that allow the extension to inject dynamic JavaScript files from a remote location. Following injection, execution continues as though the page had included it. Such content scripts provide an exceptionally powerful feature to enable a variety of malicious behaviors, as further discussed in this Section.

5.2 Affiliate Fraud

Many major merchant web sites such as `amazon.com`, `godaddy.com`, and `ebay.com` run affiliate programs that

credit affiliates with a fraction of the sales made as a result of customers referred by the affiliates. Usually merchant programs assign unique identifiers to affiliates, which affiliates then include in the URL that refers customers to the merchant site. Furthermore, affiliate programs usually associate a cookie with the user's browser so that they can attribute a sale to an affiliate within several hours after a user originally visited the merchant site with an affiliate identifier.

As an example, when a user reads product reviews on an Amazon affiliate's blog and clicks on a link to Amazon, the link includes an Amazon affiliate ID specified with the `tag` parameter in the URL, such as `http://www.amazon.com/dp/0961825170/?tag=affiliateID`. When Amazon receives this request, it returns a `Set-Cookie` header with a cookie that associates the user with the affiliate. When the customer returns to Amazon within 24 hours and makes a purchase, Amazon credits the affiliate with a small percentage of the transaction amount.

Such programs expect affiliates to bring potential customers to their sites via affiliate pages that advertise the merchant products. However, we found examples of several extensions involved in *cookie stuffing*—a technique that causes the user's browser to visit the merchant URLs without the user clicking on affiliate URLs. Doing so causes the merchant to deliver a cookie associated with the fraudulent affiliate, who then receives credit for any future, unrelated purchase made by the customer on the merchant site. Besides defrauding the merchant, the fraudulent affiliate also causes an over-write of the cookie associated with any legitimate affiliate who might have genuinely influenced the user to buy the product.

In our study, we found two kinds of extensions that defrauded affiliate programs. The first group includes extensions that provide some utility to users—such as refreshing pages automatically every few seconds, or changing the theme of popular sites like Facebook—but do not inform users of the extension author profiting from the user's web browsing. Generally, these activities involve monitoring visited URLs for merchant sites where the extension can earn a commission and modifying the outgoing requested URLs to include the affiliate ID, or by injecting `iframe`'s that include affiliate URLs.

For example, we found an extension named "*Split Screen*" (with 52K users) that allows users to show two tabs in a single window, while also stealthily monitoring the URLs visited by the user. It then silently replaces the requested URL with the affiliate's URL for sites such as `amazon.com`, `amazon.co.uk`, `hotelscombing.com`, `hostgator.com`, `godaddy.com`, and `booking.com`. For some merchants, it also sets the referrer header for outgoing requests to falsely imply a visit through the affiliate's site. The extension is able to make these changes using `tab` and `webRequest` permissions, as well as by registering callbacks on `chrome.tabs.onUpdated` to identify changes in the URL as a user types, and `chrome.webRequest.onBeforeSendHeaders` to modify the referrer header before the browser sends a request to a merchant site. We found four other extensions created by the same developer that similarly provided some small utility to the user while defrauding merchant programs in the background. Overall this developer's extensions have nearly 70K users.

Another extension we found named "Facebook Theme: Basic Minimalist Black Theme" (2.5K users) allows users to change the appearance of Facebook. Besides its stated intent, however, it also monitors browsing and appends an affiliate identifier to 7 different Amazon sites. By using its Content Security Policy (Section 2.3) to perform `eval`, it runs a highly-obfuscated hexadecimal and base64-encoded background script that stores all affiliate identifiers in Chrome's storage (using `storage` permissions), and registers callbacks on tab update events using `tab` permissions. When the user visits any URL, Chrome notifies the extension, and the extension uses regular expressions to identify target Amazon URLs for which to add an affiliate identifier. The extension then updates the URL before the browser sends the request. The creator of the extension appears well aware that the extension violates Amazon's Conditions of Use [3] and has heavily used obfuscation, evidently to evade any static analysis for detecting affiliate fraud.

As another example, we found an extension named "Page Refresh" (200 installations) that allows users to refresh tabs periodically and only requests `tabs` permission. By using the background page to listen on all tab update events, if a user visits a merchant site it sets the URL in the tab to a URL shortener that redirects the user to the same merchant page but with the affiliate identifier included in the URL, thereby stuffing a cookie into the user's browser. This extension abuses 40 different merchants, again including Amazon.

This approach has the advantage that it capitalizes on organic traffic to merchant sites, which can make fraud detection difficult because merchants see visit behavior highly similar to that they would otherwise see as a result of legitimate affiliate referrals.

The second group of extensions includes extensions that clearly state in their descriptions that the extension monetizes the user's online purchases—generally for charitable causes or donations to organizations. The intent or legitimacy of such programs is difficult to ascertain. For example, the extension "Give as you Live" [8] has over 11K users, and forms part of a larger campaign [7] to raise funds for charities from user purchases online. The extension works by adding a list of stores for which the extension author has signed up as an affil-

iate to the results of major search engines. It also adds a script on merchant sites such as `amazon.co.uk` to redirect users via its own URL. While it does bring legitimate and likely well-intentioned traffic to Amazon, the legitimate affiliates can lose out if users choose to read product reviews on affiliate sites and then make the purchase via this extension.

In fact, a plethora of extensions exists allowing users to donate to charity simply by shopping online. Another such extension uses `webRequest` permissions to modify the requested URL to the affiliate URL, including over-writing the existing affiliate URL. While this clearly constitutes cookie-stuffing, the extension advertises itself as "Help support our charity by shopping at amazon.co.uk".[3]

5.3 Information Theft

Information theft clearly reflects malicious behavior that has the potential to harm the user in a number of ways, from disclosing private information to financial loss. This broad category of abuse in many ways replicates the functionality of some malware families. Within the browser, we observe stealing of: keypresses, passwords and form data, private in-page content (e.g., bank balances), and authentication tokens such as cookies. We do not include extensions that simply re-use existing authentication tokens already present, such as extensions that spam on social networks; we discuss these in Section 5.4.

One example of keylogging we found in the Chrome Web Store, "Chrome Keylogger", is an experimental extension from researchers [14] that is now removed. Keyloggers use content scripts to register callbacks for key press events, recording the pressed key by using the messaging API to communicate with a background page. The background page then queues up data to send to a remote server. This behavior has similarities with that of extensions that steal form data, although the specific event handlers differ. Both form field theft and keylogging require the extension to specify a content script but do not require other permissions.

5.4 OSN Abuse

Online social network abuse constitutes the final category of prevalent malicious extensions we found. These extensions typically target Facebook, and spread via both the Chrome Web Store and sideloading. These extensions use existing authentication data to interact with the APIs and websites of online social networks. Previous work identified and reported Chrome extensions that abuse social networks, reporting that thousands of users had installed extensions from the Chrome Web Store that spam on Facebook [4].

```
"content_scripts": [{
  "js":["BlobBuilder.js", ... ],
  "matches":["http://*/*", "https://*/*" ],
  "run_at": "document_end"
}],
"permissions":["http://*/*", "https://*/*",
"*://*.facebook.com/",
"tabs", "cookies", "notifications",
"contextMenus", "webRequest", ...],
```

Figure 3: Permissions and content script excerpts from the manifest for an extension that spams on Facebook and creates Tumblr accounts.

We found a number of extensions that post spam messages and use other features provided by social networks, such as the ability to upload and comment on photos or query the social graph. When we execute these extensions with Hulk, the HoneyPage features allows the extensions to create elements and insert them into the DOM tree. While we do not typically inspect the visual results of our executions, in one case we observed an extension creating `div` elements to mimic Facebook status updates and inserting them into a page. The HoneyPage acted as a sink for the spam status messages resulting in a page full of spam for the infected user.

One extension of interest, *WhasApp* (a name closely resembling the popular *WhatsApp*, a mobile chat application), has since been removed from the Chrome Web Store, but we also found evidence of the same extension being sideloaded from malware. The extension targets both Facebook and Tumblr. At Facebook, the extension uploads images to Facebook and then comments on them with messages containing URLs. In some cases the links are used to spread the malicious extension to a wider audience, while other URLs sought to monetize users as part of a spam campaign to advertise products. At Tumblr, the extension creates new Tumblr accounts and verifies them in the background.

The manifest file contains permissions and content scripts that request broad access, as shown in Figure 3. The extension is in fact over-privileged, since the extension in fact does not use some of the API permissions the manifest includes. Prior work has identified over-privileging as not uncommon, even among benign extensions [13]. Figure 3 shows the extension specifically requesting access for permissions and content scripts on `facebook.com` in addition to all other sites, which provides a hint as to the sites targeted. To carry out spamming on Facebook and Tumblr account creation,

[3] The extension creator also helpfully marked the JavaScript code that adds the affiliate identifier as something to obfuscate in the future.

the extension actually only requires the use of content scripts. The abusive component of the extension is 15 lines of JavaScript that downloads a much larger remote JavaScript file containing the spamming functionality.

6 Recommendations

In this section, we frame changes to make Chrome's extension ecosystem safer. Extensions should not have the ability to manipulate browser configuration pages, such as `chrome://extensions`, that govern how users manage and uninstall extensions. Extensions should also not be allowed to uninstall other extensions unless they are from the same author or a trusted source (such as Google or Antivirus vendors). We also recommend preventing extensions from manipulating HTTP requests by **removing security-related headers** that compromise the security of web pages. This change will require modifications to several extension APIs to comprehensively address this issue, the primary one being `webRequest`.

To address cloaking and other changes in remotely included content, we suggest that Google should encourage **local inclusion of static files** in the context of a web page. Chrome supports pushing automatic updates of extensions to users, so remotely including additional JavaScript code is not necessary to support rapid changes in an extenion's code. This change will make it possible to have a more complete analysis of extension behavior, since the analysis engine—Hulk or otherwise[4]—will have the complete extension code available. To encourage developers to write completely self-contained extensions and not load additional code from the network, one could introduce a new policies, such as: if an extension loads code from a remote site, it loses permissions such as the ability to inject that new code into the visited pages.

Finally, extensions should not have the ability to **hook all keyboard events** on a given site. The *window.onkey** API that exists in JavaScript has utility for pages that want to intercept the keyboard events of their users, but in the context of extensions it provides too much power. An experimental API (*chrome.commands*) exists that allows extensions to register keyboard shortcuts; this strikes us as a step in the right direction, as this covers the common use-case for requiring access to these events.

These suggestions will not eliminate malicious extensions, but can prevent classes of attacks, and significantly facilitate the analysis of extensions.

[4] In particular, ultimately an extension store operator such as Google needs to undertake such analysis as part of its curation of the store contents.

7 Limitations

Our system uses dynamic analysis for analyzing extensions, and, as with every dynamic analysis system, the correct classification of an extension relies on triggering the malicious activity. Hulk employs HoneyPages and event handler fuzzing on the extension's web request listeners to enhance dynamic analysis, but does not provide a complete view of extension behavior. For example, we do not attempt to address cloaking that loads different code based on the client's location or time. We also will not observe behavior that depends on specific targets, such as those that require user interaction with a visited page to take effect. Similarly, pages that require sign-in pose difficulties. Hulk has a pre-set list of sites and credentials to use while visiting pages, but does not perform account creation on the fly.

Hulk's Honeypages do not currently support multi-step querying of DOM elements. While we can place elements in the DOM tree that an extension looks for, if the extension expects elements to have additional properties in order to trigger its malicious behavior, we will fail to adapt to the extension's expectations. We plan on improving HoneyPages to support multi-step querying, and for many element types and attributes this appears possible.

We currently also lack data flow analysis in the Chrome browser, a feature that would substantially improve the depth of behavior available for analysis. One example where this would prove particular useful regards keystroke interception. Without data flow tracking, we cannot automatically derive whether this information ultimately becomes transmitted to a third party via a network request.

Another difficult concern for Hulk is analysis evasion by extensions that specifically look for HoneyPages. A determined adversary with knowledge of the system could try to evade Hulk by querying for random elements in the DOM tree first, and, if found, avoid malicious activity. A similar type of evasive behavior arose for in submissions to Wepawet [17]. One way to counter this is by introducing non-deterministic HoneyPages for which DOM tree queries only succeed with a given probability. We could further enhance this approach by crawling a few million sites and building models of the existing elements to assign apt probabilities weights for different queries. This approach may also require analysis of an extension's DOM queries in case the extension repeatedly performs these in an effort to detect randomized queries. Finally, we can consider measuring code coverage to examine the impact that each DOM query has on the amount of code executed by an extension, as the extension will skip executing the malicious code when it detects the presence of an analysis system.

8 Related Work

Browser extensions have been available for Internet Explorer and Firefox for over a decade. As a result of a study of vulnerabilities in Firefox extensions, Barth et al. designed an extension architecture that promotes least privilege and isolation of components to prevent a compromised extension from gaining full access to a user's browser [5], an architecture subsequently adopted by Google Chrome. Since then, further work has examined the success of the Chrome extension architecture at preventing damage [6] and the ability of developers to correctly request privileges for their extensions [13]. Similar studies have examined the Firefox extension system to limit the potential damage arising from exploitation of extension vulnerabilities, and to improve the defenses the browser provides [27]. These works have a focus mostly tangential to our work, since the principle of least privilege does not prevent an overtly malicious extension from executing malicious code.

The security industry has documented malicious extensions in ways similar to malware reports and other new threats [2, 4]. Liu et al. examined Google Chrome extensions and, based on malicious extensions the authors built, suggested refined privileges to make detecting malicious extensions easier [21]. In our work, we build a system that performs dynamic analysis and classification of extensions, and present an analysis of malicious extensions that we found in the wild.

JavaScript-based program analysis has particular promise for benefiting our work, and in light of our current limitations we will be exploring techniques that we can adapt to improve our system's detection capabilities. Research has applied information flow analysis to Firefox extensions [10], performed taint-based tracking of untrusted data within the browser [11], used symbolic execution to detect vulnerabilities [26], applied static verification to extensions [16], contained extensions in privacy-preserving environments [20], and used supervised learning of browser memory profiles to detect privacy-sensitive events [14].

Our work has similarities to that of other malware detection and execution systems. While our implementation and requirements significantly differ from systems that execute Windows binary malware (such as Anubis [1]), at a high level we share common goals of executing and extracting data from samples. Like Anubis, Wepawet, the GQ honeyfarm, and other malware execution platforms, we share the difficult problem of triggering malicious behavior in a synthetic environment [9, 19]. Other research in this area has focused on classification and discerning malware from goodware [24].

9 Summary

In this paper we presented Hulk, a system to dynamically analyze Chrome browser extensions and identify malicious behavior. Our system monitors an extension's actions and creates a dynamic environment that adapts to an extension's needs in order to trigger the intended behavior of extensions, classifying the extension as malicious or benign accordingly. In total, we identified 130 malicious and 4,712 suspicious extensions that have up to 5.5 million browser installations, many of which remain live in the Chrome Web Store. Based on these results, we developed a detailed characterization of the malicious behavior that we found, targeted at determining the motivation behind the extension. Finally, we propose several changes for the Chrome browser ecosystem that could eliminate classes of extension-based attacks and aid with analysis.

Acknowledgments

We would like to thank our shepherd David Evans for his insightful comments and feedback. We would also like to thank Niels Provos, Adrienne Porter Felt, Nav Jagpal and the rest of the Safebrowsing team at Google for their insight and discussions throughout this project. This work was supported by the National Science Foundation under grants 0831535 and 1237265, by the Office of Naval Research (ONR) under grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, by Secure Business Austria and by generous gifts from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Anubis — Malware Analysis for Unknown Binaries. http://anubis.iseclab.org/.

[2] AMADEO, R. Adware vendors buy Chrome Extensions to send ad- and malware-filled updates. http://arstechnica.com/security/2014/01/malware-vendors-buy-chrome-extensions-to-send-adware-filled-updates/, Jan 2014.

[3] AMAZON. Associates Program Operating Agreement. https://affiliate-program.amazon.com/gp/associates/agreement/, 2012.

[4] ASSOLINI, F. Think twice before installing Chrome extensions. http://www.securelist.com/en/blog/208193414/Think_twice_before_installing_Chrome_extensions, Mar 2012.

[5] BARTH, A., FELT, A. P., SAXENA, P., AND BOODMAN, A. Protecting Browsers from Extension Vulnerabilities.

In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2010).

[6] CARLINI, N., FELT, A. P., AND WAGNER, D. An Evaluation of the Google Chrome Extension Security Architecture. In *Proceedings of the USENIX Security Symposium* (2012).

[7] CHARLES ARTHUR. Infographic: Internet shopping. `http://www.theguardian.com/technology/blog/2011/jul/04/internet-shopping-infographic-give-as-you-live-charity`, 2011.

[8] CHROME WEB STORE. Give as you Live. `https://chrome.google.com/webstore/detail/give-as-you-live/fceblikkhnkbdimejiaapjnijnfegnii`, 2013.

[9] COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the World Wide Web Conference (WWW)* (2010).

[10] DHAWAN, M., AND GANAPATHY, V. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009).

[11] DJERIC, V., AND GOEL, A. Securing script-based extensibility in web browsers. In *Proceedings of the USENIX Security Symposium* (2010).

[12] F-SECURE. Coremex innovates search engine hijacking. `http://www.f-secure.com/weblog/archives/00002689.html`, April 2014.

[13] FELT, A. P., GREENWOOD, K., AND WAGNER, D. The Effectiveness of Application Permissions. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)* (2011).

[14] GIUFFRIDA, C., ORTOLANI, S., AND CRISPO, B. Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2012), ACM.

[15] GOOGLE. What are extensions? `https://developer.chrome.com/extensions/index`, 2014.

[16] GUHA, A., FREDRIKSON, M., LIVSHITS, B., AND SWAMY, N. Verified Security for Browser Extensions. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 115–130.

[17] KAPRAVELOS, A., SHOSHITAISHVILI, Y., COVA, M., KRUEGEL, C., AND VIGNA, G. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the USENIX Security Symposium* (2013).

[18] KAY, E. Protecting Chrome users from malicious extensions. `http://chrome.blogspot.com/2014/05/protecting-chrome-users-from-malicious.html`, May 2014.

[19] KREIBICH, C., WEAVER, N., KANICH, C., CUI, W., AND PAXSON, V. GQ: Practical containment for measuring modern malware systems. In *Proceedings of the ACM Internet Measurement Conference (IMC)* (2011), ACM, pp. 397–412.

[20] LI, Z., WANG, X., AND CHOI, J. Y. Spyshield: Preserving privacy from spy add-ons. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)* (2007).

[21] LIU, L., ZHANG, X., YAN, G., AND CHEN, S. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2012).

[22] LUDWIG, P. No more silent extension installs. `http://blog.chromium.org/2012/12/no-more-silent-extension-installs.html`, Dec 2012.

[23] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).

[24] RAJAB, M. A., BALLARD, L., LUTZ, N., MAVROMMATIS, P., AND PROVOS, N. CAMP: Content-Agnostic Malware Protection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2013).

[25] REDDIT. Reddit: I am One of the Developers of a Popular Chrome Extension.... `http://www.reddit.com/r/IAmA/comments/1vjj51/i_am_one_of_the_developers_of_a_popular_chrome/`, Jan 2014.

[26] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[27] TER LOUW, M., LIM, J. S., AND VENKATAKRISHNAN, V. Enhancing Web Browser Security Against Malware Extensions. *Journal in Computer Virology 4*, 3 (2008), 179–195.

[28] WANG, D., SAVAGE, S., AND VOELKER, G. M. Cloak and Dagger: Dynamics of Web Search Cloaking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011), ACM, pp. 477–490.

[29] WEST, M. An Introduction to Content Security Policy. `http://www.html5rocks.com/en/tutorials/security/content-security-policy/`, 2012.